Universidade de Coimbra
Departamento de Engenharia Informática
Sistemas Distribuídos 2018/19

Exercises #1 — Thread Programming in Java

All distributed systems are built using multiple threads or processes. For example, a server must handle many clients and one alternative is to run one thread for each client. Therefore, multi-threaded programming in Java will be used throughout the course.

1. Run the program `MultiThreadDemo.java`. This program demonstrates one of two ways of creating threads (in the other alternative, a class extends the Thread class directly). Students can experiment several values for the sleep time in the main thread and verify that the program finishes only after termination of the last of the four.

2. The program `DemoJoin.java` shows how to wait for the termination of a thread. Among other things, understand the need to handle the `InterruptedException`.

3. Take a look at the programs `Synch*.java`. The output should be a word enclosed in square brackets per line. `Synch.java` fails to accomplish that, because it has no synchronization among threads. `Synch1.java` and `Synch2.java` show two different ways, using the synchronized keyword to solve the problem. `Synch3.java` fails on the task. You should try to understand yourself why that happens. Also note that there is no guarantee on the order lines show up on the output.

4. The program `PC_wrong.java` clearly demonstrates that the synchronized keyword does not solve all the synchronization problems. It cannot ensure that the two threads alternate in the access to the queue. Two approaches are possible (in fact, there are more): `PC_naive.java` uses busy waits, which are usually discouraged, because it wastes resources. `PC_ok.java` solves the problem correctly using monitors. You should ask yourself the following questions to check whether you understand the program correctly:

- When a thread does a `wait()` on an object does it release the corresponding lock?
- What is the goal of the variable `valueSet`, and why should it be declared `volatile`?
- If no thread is blocked in a `wait()` what happens to a `notify()` sent by another thread?
- Does this program work for more than one producer and more than one consumer?

5. Is it possible to write a `Semaphore` class, with the following methods, using monitors? Try it yourself.

- `Semaphore(int val);`
- `doWait();`
- `doSignal();`

What does this tell us about the relative power of both synchronization mechanisms (semaphores *vs.* monitors)?